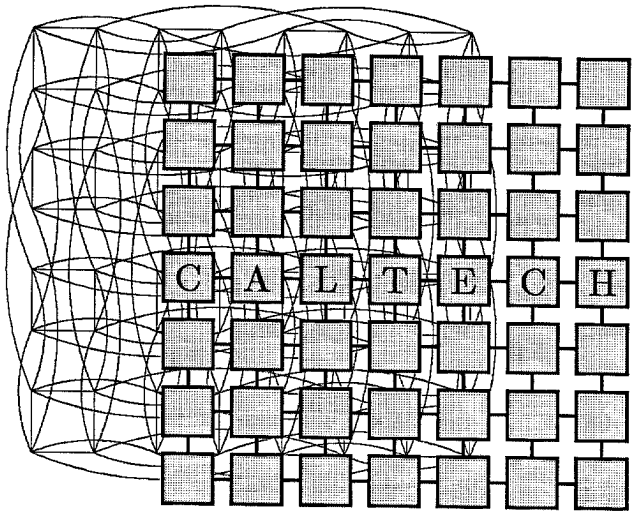

SUBMICRON SYSTEMS ARCHITECTURE PROJECT

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Semiannual Technical Report

Caltech Computer Science Technical Report

Caltech-CS-TR-88-5

7 April 1988

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-88-5

7 April 1988

Reporting Period:	1 November 1987 – 31 March 1988
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	William C. Athas
	K. Mani Chandy
	Alain J. Martin
	Martin Rem
	Charles L. Seitz

Sponsored by the
Defense Advanced Research Projects Agency
DARPA Order Number 6202

Monitored by the
Office of Naval Research
Contract Number N00014-87-K-0745

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the five-month period, 1 November 1987 to 31 March 1988, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental message-passing concurrent computers, and includes related efforts in concurrent computation and VLSI design.

1.3 Highlights

Some highlights of the previous five months are:

- The Ametek Series 2010, a second-generation medium-grain multicomputer developed as a joint project between our research project and Ametek Computer Research Division, was announced as a commercial product. A 16-node engineering prototype has been demonstrated running numerous application programs. (See section 2.1 and the paper "The Architecture and Programming of the Ametek Series 2010 Multicomputer" in the appendix.)
- Enhancements to the Cantor programming system (section 3.1).
- Reference definition of the functions of the Cosmic Environment and Reactive Kernel (sections 3.2 and 3.3).
- High-quality self-timed VLSI designs are being produced by a compilation procedure that is now fully automatic (sections 4.1 and 4.2).
- Fast "Mesh Routing Chips" (section 4.5).

2. Architecture Experiments

2.1 Second-Generation Medium-Grain Multicomputers*

Chuck Seitz, Alain Martin, Bill Athas, Charles Flaig, Jakov Seizovic, Craig Steele, Wen-King Su

On 19 January 1988, the Ametek Series 2010 multicomputer was announced at the 1988 Hypercube Conference in an invited talk by Chuck Seitz. This is the first multicomputer to reach our goal for the second generation of multicomputers of a 100× improvement over the first-generation hypercube multicomputers in the relationship between communication and computing performance. A paper on "The Architecture and Programming of the Ametek Series 2010 Multicomputer," to appear in the proceedings of the 1988 Hypercube Conference, is included as an appendix to this report.

In this same week, a 16-node engineering prototype of the Ametek Series 2010 was demonstrated and benchmarked running application programs. All of these programs had been developed and run previously on Cosmic Cubes, Intel iPSC/1s, or "ghost cubes." In all cases, the programs ran correctly on the Ametek Series 2010, requiring only compilation and linking with the appropriate compatibility libraries. In March 1988, a 16-node system with 20 Mflop vector floating-point accelerators on each node was demonstrated running an edge-detection benchmark at 170 Mflops. Systems at the centerline design point of $N = 256$ nodes will be capable of a peak performance of 1 GIPS, 5 Gflops, and 5 Gb/s network bilateral bisection bandwidth.

The announcement and demonstration of the Ametek Series 2010 was the culmination of a 16-month joint development program with Ametek Computer Research Division. Our Caltech project provided the architectural design, routing chip designs and prototypes, and system software consisting of the Reactive Kernel (RK) node operating system and the Cosmic Environment (CE) host runtime system. Ametek provided the detail logical designs, physical designs, parts, assembly, and construction of the prototypes to our specifications and designs. Ametek also ported RK, and wrote the necessary interface routines to CE.

Considering the complexity of this project (new architecture, new system software, new custom mesh routing chips, new node design, new host interface, and new packaging), it proceeded very smoothly. The RK port required only two months for the Ametek system-programming team, and about 90% of the resulting system is identical to C source code provided by Caltech. The only serious problem that occurred in the entire project was routing chips that did not function correctly

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Ametek Computer Research Division (Monrovia, California).

on first silicon. This problem was traced to a missing contact cut and mistake in the signal naming that did not allow this error to be detected in the usual extraction and switch-level simulation process. The second-pass silicon on this self-timed SCMOS chip, one of two independent mesh routing chip designs, functioned correctly. The other design worked correctly on first silicon.

Ametek has non-exclusive licenses to Caltech patents on the Cosmic Cube architecture and message-passing mechanisms, to Caltech patents on mesh routing chip organization, and for Caltech system software. As part of this license arrangement, Ametek will be contributing a 256-node system to Caltech. An allocation of cycles on this system will be made available to guest researchers, as is currently done with our Cosmic Cubes and iPSC/1.

2.2 Mosaic Project

Bill Athas, Charles Flaig, Glenn Lewis, Don Speck, Wen-King Su, Chuck Seitz

The Mosaic C is a message-passing MIMD multicomputer with single-chip nodes. The stipulation that the nodes fit on a single chip limits the storage for each node, so that relatively fine-grain concurrent programming techniques must be used. We are working toward building a 16K-node Mosaic system using nodes fabricated in 1.2μm CMOS technology, with a near-term milestone of a 1K-node system using nodes fabricated in 2μm CMOS.

The status of the Mosaic C chip design is described in section 4.4, and the current work on the Cantor programming system that we shall use for programming the Mosaic is described in section 3.1.

2.3 Cosmic Cube Project

Bill Athas, Michael Lichter, Wen-King Su, Jakov Seizovic, Chuck Seitz

This section summarizes the current usage and the hardware and software status of our first-generation multicomputers, the Cosmic Cubes and Intel iPSC/1 d7. These systems continue to operate reliably. The major system software changes introduced in the fall 1987 have caused no significant problems, and have improved the compatibility between the Cosmic Cubes, iPSC/1, "ghost cubes," and the Ametek Series 2010.

Overall usage has been moderately heavy. The most time-consuming application in this period from within our own group have been an extensive series of simulations by John Ngai concerned with the maximal utilization of networks with faulty routers or channels (see section 4.6). Supersonic flow computations being performed by students and faculty in Aeronautics at Caltech continue as the largest share of outside use. Other guest users include David Mizell's group at ISI, who have been experimenting with distributed simulations, and several researchers doing neural network simulations.

Neither the 64-node nor 8-node Cosmic Cubes has exhibited a hard failure in this five-month period. These cubes have now logged 3.2 million node-hours with only three hard failures. The calculated node MTBF of 100,000 hours reported before these machines were constructed was extremely conservative. A node MTBF in excess of 1,000,000 hours is probable, and can be stated at a 50% confidence level.

Our Intel iPSC/1 d7 (128 nodes) was contributed to the Submicron Systems Architecture Project as a part of the license agreement between the Caltech and Intel, and is accessible via the ARPAnet to other DARPA researchers who may wish to experiment with it. To request an account, please contact `chuck@vlsi.caltech.edu`. Delivery of the alpha test unit of the new Ametek Series 2010 system is anticipated in about two months. This system will be available for outside use on a similar basis.

3. Concurrent Computation

3.1 Cantor

Bill Athas, Nanette Jackson, Chuck Seitz

Continuing research using the Cantor programming system has focused on writing application programs, and on refining the Cantor programming model. Our goal in writing application programs is to develop programs that are suitable for execution upon fine-grain multicomputers, such as the Mosaic C. Our experience from writing programs in Cantor is used to refine the Cantor language definition, and the instrumentation of these programs has provided the essential parameters for the design both of the Mosaic C and of an experimental Cantor Engine.

New applications programs written in Cantor include a program to enumerate paraffin isomer molecules, a program to test for graph isomorphism, and a program to analyze a chessboard for a checkmate configuration and report the possible moves to escape checkmate. This latter program is over 750 lines of code.

From these programs, plus the programs previously reported, we have observed three general paradigms for writing concurrent programs in Cantor.

1. The first paradigm is the transformation of functional or dataflow programs into Cantor programs. The transformations applied are systematic and the application of continuations and futures is straightforward.

2. The second paradigm is the transformation of a program specification into a Cantor program. The typical problems from this area are combinatorial searches using a breadth-first or divide-and-conquer approach. However, the resulting Cantor object graphs are not trees but are series-parallel (S-P) graphs. The S-P graphs are formed from the factoring of recursions into two parts: the invocation of the recursive call, and the rendezvous with the return from the recursion.

3. The third paradigm, and by far the most interesting, is the object program as an apparatus for performing a computation. A simple example is the wheel-driven prime sieve, in which the computation is represented by a number generator called the wheel and the infinite sieve. More interesting examples are simulation in which each object in the simulation is represented by a Cantor object.

Our latest revision of Cantor is version 2.2. This version supports dynamically-allocated vectors and functional abstraction. Cantor 2.1 supported vectors in which the size of the vector was computed at compile time. This restriction supported efficient compilation of vectors, but was of limited usefulness. We often found that vectors are combined into larger vectors, in which the size of the component vectors is data-dependent. Thus, Cantor 2.2 supports vectors that are allocated on demand.

Cantor 2.2 also provides for functional abstraction over expressions. Previously, functional application was used to produce future reference values for new objects. Functions can now produce a value of any type. The invocation of a function is quite similar to creating a new object. The function is expected to produce a list of return values for the caller. The list of return values is passed back to the caller by message-passing. In the interim between calling a function and receiving the list of return values, the caller leaves the running state. All messages received between calling a function and receiving the reply message are enqueued. Once the reply message returns, the object is again a candidate for execution, and all messages that were enqueued are processed using the normal execution rules. Because calling a function causes the caller to leave the running state, the context for the caller must first be saved. The saving of context is performed by the compiler using live-variable analysis.

Our next refinement for the Cantor programming system is to provide a facility for supporting custom objects and functions, namely, machine code that has been separately prepared, but which is compatible with the Cantor execution model. Our plan for incorporating custom objects and functions into Cantor are to provide for separate compilation of Cantor object definitions and functions, and then link the native definitions with the definitions for custom objects and custom functions.

The latest stable and distributed version of Cantor is 2.0. It is expected that version 2.2 will become available for distribution to other research groups in mid-summer.

3.2 The Cosmic Environment

Wen-King Su, Chuck Seitz

The Cosmic Environment (CE), our generic, portable multicomputer interface, has been augmented with the Unix standard IO libraries. This new feature was made possible by the addition of RPC messages. A RPC message is identical to a normal message, with the exception that a program has the option of selectively waiting for a reply message. When a program issues an `xrecvrpc`, the program is blocked until a RPC message is received. The message is then returned to the process.

The "ghost cube," a multicomputer simulator that is made of a group of NFS-connected UNIX computers or workstations, has proved to be very popular. Ghost cubes now have a hook for running the debugger program. Users can run dbx on their node programs and test their programs fully on a ghost cube before moving them unmodified to a real multicomputer. The Cosmic Environment now supports the original Cosmic Cubes, the Intel iPSC/1, ghost cubes, and the new Ametek Series 2010.

The documentation for CE version 7.2 and for the Reactive Kernel is now completely up-to-date. The latest edition of "The C Programmer's Abbreviated

Guide to Multicomputer Programming" (Caltech-CS-TR-88-1) was completed in January 1988, and 300 copies were distributed to our user community. CE has now been distributed to well over 100 sites in the United States, Canada, Western Europe, Scandinavia, and Israel.

3.3 The Reactive Kernel

Jakov Seizovic, Chuck Seitz

The Reactive Kernel (RK) has been successfully ported to one of the second-generation machines, the Ametek 2010, and has been running reliably on that system for the past three months. This uneventful port has demonstrated that our goal of making RK highly portable was achieved. The careful layering of the RK structure, with well-defined interfaces between the layers, has enabled the testing and tuning RK by incrementally adding more complex features, without interfering with the already tested ones. Much of this activity has been concerned with trying to get as much performance as possible out of the message system. The following back-reference problem is an example of this kind of tuning.

Consider the following program fragment, which occurs frequently in programs with the reactive primitives:

```
p = xmalloc (length);
build_the_message (p);
xsend (p,node,pid);
```

At the allocation time, a data structure called a *descriptor*, which contains the relevant information about the allocated block, is associated with the block. This scheme creates a *back-reference* problem; that is, a problem of finding the appropriate descriptor given the pointer to a particular memory block.

An obvious solution is to keep the descriptor pointer, or the whole descriptor, within the memory block itself. However, this solution is not satisfactory, because misuse or overwriting these pointers or descriptor by user processes can cause an operating system error. What we need is a dictionary, a set representation with the *insert*, *delete*, and *member* operations. The set elements are descriptors, and the *keys* are pointers to the memory blocks. The algorithm used in RK makes a compromise between the time and space complexity. The idea of the algorithm is as follows: in order to access an element of the set, we perform a search along an N-ary tree for k steps, whereby with each step we reduce the number of possible elements by a factor of N. After k steps we are left with at most $n = N_{max} N^k$ possible outcomes, and can resolve the remaining ambiguity, if any, by a sequential search.

Given the size of the memory used for messages, the average number of messages in the memory, the distribution of message sizes, and the cost function representing the balance between the memory utilization and the time required to access an

element of the set, we are able to find an optimal configuration. Since the parts of the data structure are dynamically allocated, it is even possible to change the configuration 'on-the-fly,' after obtaining the information about the current message traffic. If the reconfiguration is performed at the point when there are no messages in the system, it can be done with essentially zero cost.

The only important addition to RK functions that we are planning is a variant of the standard spawn function that places a process automatically. Associated with this addition will be an improved mechanism to cache process code, so that the speed of spawning a new process will be comparable to that of message passing. This addition is part of our long-term plan to make the semantics of a subset of RK message and spawning functions identical to those of Cantor (section 3.1).

3.4 Concise — A Concurrent Circuit Simulator*

Sven Mattisson, Lena Peterson, Chuck Seitz

The concurrent circuit simulation program, Concise, currently runs under the Cosmic Environment with the reactive primitives on UNIX computers; on all forms of multicomputers, including ghost cubes; and also on a Sequent under the Cosmic Environment.

Experimental modifications have been made over the past several months in order to make clustering of tightly coupled circuit nodes possible. The clustered "difficult" nodes are solved by a direct method, thus increasing the convergence rate for many circuits, both digital and analog. An investigation of automatic circuit partitioning methods is currently underway.

In another effort, Concise has been used by Anthony Skjellum in the Chemical Engineering Department at Caltech for the simulation of distillation columns. This work has shown that it is possible to use Concise to simulate dynamic systems that are not at all like circuits. As part of this effort, Concise has been modified to make it easier to install models of other kinds of "devices."

This work on Concise will be presented in two papers at the IEEE International Symposium on Circuits and Systems (ISCAS) in Helsinki, June 1988.

3.5 Transformational Derivation of Distributed Algorithms

Kevin S. Van Horn, Alain Martin

In the past several months we have begun to develop a transformational method for deriving concurrent programs, with an emphasis on the derivation of distributed programs. A transformational derivation of a concurrent program proceeds as

* This segment of our research is a joint project between the Caltech Submicron Systems Architecture Project and the Department of Applied Electronics at the University of Lund, Sweden.

follows. Given a problem to solve, one first produces a simple, easily-understood program with a straightforward correctness proof. This program may be inefficient, involve globally shared variables, make no use of message-passing, and may not even have any explicit concurrency. One then applies a series of transformations to this program, proving any conditions which must hold for the transformation to be valid, until one obtains an efficient distributed program.

There are several advantages to such a method. One is that the conceptual structure of the algorithm becomes much clearer. The original program expresses the essence of the algorithm, which is elaborated by succeeding transformations that, for example, implement global tests and updates of global variables, and detect termination. Another advantage is that the correctness proof of the final program is broken into smaller, more easily managed pieces. Perhaps the biggest advantage is that it allows one to work out and prove correct an intermediate solution to the problem before deciding on many details of the final algorithm.

The notation used is a variant of Chandy and Misra's UNITY. We are at present restricting ourselves to terminating programs in order to avoid some thorny issues that arise with non-terminating programs, although it appears that many of the transformation techniques developed so far should be applicable to both. A program in this notation consists of a declaration of variables with their initial values, a set of assignments, a termination condition, and a result expression. The operation of such a program can be described informally as follows: repeatedly (and fairly) choose an assignment or the termination condition; if an assignment is chosen then execute it, otherwise evaluate the termination condition and if it holds then terminate, returning the value of the result expression in the present state. The kinds of transformations we apply to these programs include data refinement, distributing and/or combining assignments, superposing new variables, removing superfluous variables, and strengthening the termination condition.

This transformational method has been used to derive a number of algorithms, some original and some preexisting. These include a distributed best-first search algorithm, various all-points shortest path algorithms, two termination-detection algorithms, and a distributed minimal spanning tree algorithm that appears to be a significant improvement over that of Gallager *et al.*

3.6 A Multicomputer Z–Buffer Program

Glenn M. Lewis, Wen-King Su, Chuck Seitz

As a demonstration program for multicomputers running the Reactive Kernel, we have written a distributed version of the usual graphics Z–buffer program. It takes input from any graphics rendering program that generates three–dimensional coordinates and color, and sorts the information such that the result simulates a true hidden-line representation of the image.

4. VLSI Design

4.1 Standard-cell Placement and Routing Program

Steve Burns, Pieter Hazewindus, Alain Martin

To facilitate rapid layout of chips, we have designed a new placement and routing program, `gladys`. This program takes as input a circuit description consisting of a set of gates, which may be generated by the circuit compiler. This description is then converted into a standard-cell layout. The result is a number of *towers* of standard cells, with routing channels in between towers. In the standard cells, no metal2 is used, so that the router can route between towers over standard cells.

The program consists of a placement algorithm, which attempts to reduce wire lengths by simulated annealing. Thereafter, global routing is done to route between cells in non-adjacent towers, and finally, a channel router does local routing in the channels between towers, using a greedy three-layer routing algorithm. The router has no global considerations when deciding on the location of wires; hence, the algorithm is very fast (it typically routes a medium-sized chip in a matter of seconds).

We have compared this algorithm with layouts generated by MOSIS's FUSION tool. The FUSION layout is about 50% larger if no placement is specified, and about 10% larger if it is supplied with the result generated by the previously mentioned placement algorithm. We expect to be able to reduce our layout size by 5–10% by using a better channel routing scheme, and by incorporating some global optimizations.

As a final step in the automatic transformation of a program into a chip, a padrouter needs to be constructed.

4.2 Bit-serial Routing Chip Compiled from a High-level Description

Steve Burns, Alain Martin

We have designed and fabricated a self-timed bit-serial routing chip compiled directly from a program. All stages of the compilation were performed automatically, using a procedure with the following structure:

(i) parse tree generation,

(ii) tree-based (global) optimization,

(iii) operator generation,

(iv) peephole (local) optimization,

(v) operator to standard-cell binding,

(vi) standard-cell placement, and

(vii) inter-cell routing.

Stages (i) through (v) were performed using a PROLOG-based 'CSP to Self-timed Circuit' compiler hinted at in the last semiannual DARPA report, and described in more detail at the 1988 MIT VLSI Conference. The placement and routing steps were performed by the MOSIS FUSION system.

The "Compiled MRC" was tested and functions correctly with a throughput of 5.6 MHz (four-phase handshake in 180 ns). The latency through a single router element is 253 ns. The performance of this chip is somewhat disappointing, caused mostly by an inadequate implementation of step (v). A more careful implementation of the 'operator to standard-cell binding' step should increase the performance of the compiled chips by a factor of two.

Global optimizations will also improve the circuits produced by this compilation method. In particular, reshuffling of communication actions will, in many cases, produce more efficient (in terms of area and speed) implementations. However, in general, reshuffling introduces deadlock. Global analysis of the system is necessary to show that reshuffling will not introduce deadlock. Currently, this global analysis is performed manually, and annotations are added to the source programs specifying when the communications may be interleaved. We are working to automate this analysis. The "Compiled MRC" included a router element with reshuffled communications. The throughput of the reshuffled router increased 20% to 6.7 MHz (four-phase handshake in 148 ns). The latency was reduced more dramatically to 81 ns.

4.3 Characterization of Communication Patterns with Constant Response Time

Tony Lee, Alain Martin

In a system of identical communicating processes connected in a regular structure (linear array or mesh) —such systems are usually called systolic arrays—, the order of communications of a process with its neighbors can be modified to improve performance. However, it is, in general, difficult to predict the effect of such a reordering: it may cause deadlock, or it may lead to a behavior where the "response-time" of a process to a communication depends on the number of processes in the systems.

It so happens that the reshuffling of actions in a handshaking expansion that we perform during the compilation of a communicating process into self-timed circuits have the same properties: although they are introduced to improve preformance, they may lead to deadlock or to a variable response-time.

We have defined a necessary and sufficient condition for a communication pattern in a linear array to be deadlock-free and to have a constant response-time.

4.4 Mosaic Elements

Bill Athas, Charles Flaig, Glenn Lewis, Don Speck, Wen-King Su, Chuck Seitz

The Mosaic C chip is composed of three main parts: RAM & ROM, channels, and processor. Our strategy for verifying the design of this very complex chip and characterizing its yield on MOSIS runs is initially to fabricate and test the three main parts separately. After the parts have been well characterized, their layouts will be combined onto a single chip. All the sections except for the ROM have been designed and layed out. The RAM and channels sections have been fabricated and verified. The final assembly of the processor and of the entire chip are expected to be accomplished this summer.

The target technology for the Mosaic C is MOSIS SCMOS with $0.6\mu m \leq \lambda \leq 1.5\mu m$. Target maximum chip size is $36mm^2$, or $100M\lambda^2$ with $\lambda = 0.6\mu m$, and $16M\lambda^2$ with $\lambda = 1.5\mu m$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.

The architecture of the Mosaic C and the design of the Mosaic C chip are described in previous semiannual technical reports.

4.4.1 Mosaic C dRAM

Our basic strategy has been to develop a 4-transistor dRAM that is a low-risk design with a relatively large area, and a 2-transistor dRAM that is a higher-risk design but has a relatively small area. The following efforts have been aimed at improvements in the 4T dRAM:

Decoders: Due to pitch constraints, the RAM and ROM row select decoders must be precharged. Our desire to charge and discharge as few decoder outputs as possible leads us to domino NAND gates.

However, precharging through a series transistor chain can be very slow. Because the transistors are turning off as charge is drained, the precharge time (and hence the input setup time) is cubic in the chain length. The setup time allowance for the decoder is zero, so each internal node must have its own precharger. To make room for those prechargers, series chains must be coalesced into trees, with a branching width limited to 70λ so that internal nodes remain accessible and stay small enough to not need area-consuming metal strapping.

For speed it's conventional to predecode bit pairs so that fewer series transistors are needed. However, with the tapering transistor sizes afforded by the tree structure, the time saved by removing half of the transistors does not recoup the predecode overhead. Predecoding only gains speed if applied just to the leaves of the trees, where the predecode time is not in the critical path.

RAM simulation: We have discovered a bug in SPICE2G.6 which greatly overestimates the effective gate capacitance of pass transistors. When a pass

transistor is cut off by back-gate bias, the CMEYER routine calculates full gate capacitance, as if the MOS capacitor were in accumulation, when it should be in deep depletion with a much lower capacitance.

SPICE2G.6 also neglects the channel-to-bulk capacitance, though that bug at least has an easy workaround (increase the source area by the amount of gate area).

4.4.2 Channels

The width of the channels in our current designs has been increased to 4 bits. The registers and bus drivers for the processor interface have been completed, and state tables for the control circuitry have gone through a first draft.

4.4.3 Processor

The Mosaic C processor datapath design and layout is complete, and it simulates correctly with MOSSIM. Our efforts of the past several months have included continued checking of the microcode, and attempts to improve the speed of the control PLA.

4.5 Self-Timed Mesh Routing Chips

Charles Flaig, Chuck Seitz

Samples of the Mesh Routing Chips (MRCs) sent to MOSIS for fabrication in September were received and tested in December, and functioned correctly. The 95% yield was excellent, but the speed was below expectations. The cycle times for these chips was about 100ns in 3μm SCMOS, which was a factor of two less than expected. The fallthrough time for each FIFO stage was also high, at about 15-20ns. A large part of the problem was traced to long wires in the 2/4-cycle conversion circuitry. A design oversight placed excessive capacitive loads on relatively weak transistors. There were also some "hurry up" design shortcuts that were detrimental to the speed. Based on experience with another MRC design, this design would have exhibited a satisfactory cycle time of about 33ns in 1.6μm SCMOS, but our studies of the internal timing of this chip revealed a way to increase the speed quite dramatically.

A new version of the MRC was begun in December. This design corrects all of the known problems and shortcuts in the original MRC, but also implements the FIFOs and internal switching with a more efficient signaling scheme. The external signaling conventions must still conform to the MRC specification. The major internal changes are as follows:

1. New FIFO stages. The previous FIFOs used interconnected C-elements which would store a flit (flow control unit) in two successive stages. The new FIFOs use some additional state and timing information to produce a load pulse of fixed

width, and thus store a flit in a single stage. While this does not significantly affect the fallthrough or cycle time, it increases the amount of storage available for blocked packets by a factor of two.

2. New 2/4-cycle converters. The fixed width load pulse produced by the new FIFOs allowed the construction of a simplified, and much faster, 2/4-cycle conversion circuit for an interface to the external 2-cycle request/acknowledge signaling. This conversion circuit also introduces a limitation on the minimum cycle time for the output of a channel, which we must balance with an internal delay on the output request driving logic.

3. An improved decrementer. The old decrementer had badly sized transistors which resulted in very poor performance for decrementing large numbers.

4. Improved topology to minimize the length and capacitance of connecting wires, as well as to eliminate the need for any wasteful "padding" space previously needed to compose all the cells. As a result, the new MRC core is about 20% smaller.

SPICE simulations showed that the new MRC should indeed have much better performance than the original. To get a solid test of the new FIFO and 2/4-cycle converter stages, a 64-stage FIFO was constructed and sent out for fabrication in 3μm SCMOS at the end of January.

This FIFO returned early in April, and was promptly tested. The new FIFO fallthrough time is about 7.7ns, an improvement (same technology) of a factor of two over the original MRC. The Request$\rightarrow$Acknowledge cycle time is about 10ns, giving an overall cycle time of about 20ns (50M flits/s). This is five times faster than the original MRC, and exceeded our expectations by a factor of two! In a complete MRC, rather than just a simple FIFO, there will be longer wires and larger loads, but many circuits have also been tweaked slightly, so it should also have a 20ns cycle time. Fabrication in a 1.6μm feature size usually triples the speed of circuits, but in this case the cycle time will clearly be limited by the inductance of the chip leads. A low-inductance package will be critical for realizing the exceptional performance that the chip itself can deliver. We are expecting these designs to achieve a throughput well in excess of 100M flits/s.

All of the cells have now been laid out and individually simulated for the new MRC. A few simulations of compositions have to be performed next to try to minimize the internal cycle time. Then the complete MRC can be composed and switch-level simulated using Mossim and the AutoMossim driver program. It is expected that this can be done by late April and, barring unexpected problems, we should be able to send it to fabrication by early May.

If successful, we expect this new MRC chip to replace the MRC currently used in the Ametek Series 2010 multicomputer. With help from George Lewicki, this

design will also be transferred to an Intel fabrication process for use in a future Intel multicomputer.

4.6 Adaptive Routing in Multicomputer Networks

John Y. Ngai, Chuck Seitz

We continue to investigate the use of adaptive routing techniques to improve and sustain the performance of multicomputer communication networks. We have found what we believe is a scheme that is simple enough to be realizable in practice, and that outperforms even the highly evolved oblivious wormhole routing schemes. Completion of this work and publication of Ngai's thesis is expected in the next six months.

Our efforts have been divided into three different areas relating to three different aspects of the Adaptive Cut Through (ACT) routing technique:

(1) *Performance Analysis and Simulations:* Extensive simulations of various traffic patterns have been conducted. Some of the preliminary results were summarized in the last semiannual report. A detailed summary will appear in the dissertation.

(2) *Trial Implementation:* Here efforts are focused in isolating and understanding the major design trade-offs involved in a practical implementation of the ACT router. The investigation is conducted as a student group design project in the VLSI design class, with crucial contributions also from Charles Flaig and Glenn Lewis.

(3) *Reliability Enhancement Studies.* The single most important aspect of the routing formulation is its capability to exploit the existence of multiple paths intrinsic in most of the richly connected multicomputer networks. In addition to potential performance improvements, here our efforts are to investigate and evaluate the potential reliability enhancements that can be achieved. In particular, motivated by the desire to build high-performance networks through hardware realization of the routing operations, we look for the solution which allows us to continue the use of the original hardware routers systematically with little or no change in the routing hardware. To this end, we have developed a simple framework based on convexity and reachability defined with respect to the original routing relations. Extensive computations and simulations are conducted, with the result that the loss of a few percent of the routers or nodes will still allow well in excess of 80% of a multicomputer to remain in service.

4.7 The Notorious CIF-flogger Program

Glenn Lewis, Chuck Seitz

The CIF-flogger is a multicomputer program for flattening CIF files, rasterizing the geometry, and for performing parallel operations on the geometry in stripes. It runs under the CE/RK system, and hence, on most available multicomputers, including the Ametek Series 2010.

CIF-flogger currently supports simple bloat, shrink, and logical operations on the flattened geometry, and hence can perform most geometrical design-rule checks. It will eventually provide complete design-rule checking, well checks, and circuit extraction. Based on timings on the iPSC/1, CIF-flogger is expected to perform design rule checks for 100K-transistor chips in much less than 1s per rule on second-generation multicomputers.

4.8 Pads and Pad Frame Generation

Charles Flaig, Glenn Lewis, Chuck Seitz

Motivated in large part by the variety of mesh routing chips (MRCs) being designed, a similarly large variety of new pad circuits have been designed for $\lambda = 0.6\mu$m, 0.8μm, and 1.0μm MOSIS SCMOS processes. The unusual features of these designs include:

1. The use of longitudinal (bipolar) clamp transistors for static and overvoltage protection. These protection circuits appear to be very effective.

2. Experimental use of pad spacings that are less than the standard MOSIS 200μm pad pitch. MRCs run with $\lambda = 0.8\mu$m have used a $191\lambda = 152.8\mu$m= 6.02mil pad pitch with 33 pads per edge. When one run of 50 chips was bonded in the standard MOSIS 132-pin PGA package* (small package well variety), we observed 83% yield on this MRC overall, and 100% bonding yield. Output edge times were less than 2ns, and these (self-timed) MRCs operate at about 30Mflits/s.

Our thanks to George Lewicki at MOSIS for tolerating and perhaps even encouraging these experiments.

These efforts, and related efforts in helping MOSIS with standard frames, have required the generation of many pad frames. Thus, the pad library was created along with some tools that have automated generation of pad frames, and have saved countless hours of tedious work.

* Other users of the MOSIS 132PGA packages are advised to study the documentation on this very nice (as PGAs go) package, noting in particular that 12 of the pins have about 5× lower resistance and inductance than the rest. We have used these pins for Vdd and GND.

4.9 SunCIFP

Glenn Lewis, Wen-King Su, Chuck Seitz

A new version of the CIFP program has been written and is available for distribution. It runs on Sun workstations, and creates a display of CIF geometry on a Sun window.

To be published in the Proceedings of the 1988 Hypercube Conference

The Architecture and Programming of the Ametek Series 2010 Multicomputer

Charles L. Seitz, William C. Athas, Charles M. Flaig,
Alain J. Martin, Jakov Seizovic, Craig S. Steele, Wen-King Su

Department of Computer Science
California Institute of Technology

Background

During the period following the completion of the Cosmic Cube experiment [1], and while commercial descendants of this first-generation multicomputer (message-passing concurrent computer) were spreading through a community that includes many of the attendees of this conference, members of our research group were developing a set of ideas about the physical design and programming for the second generation of medium-grain multicomputers.

Our principal goal was to improve by as much as two orders of magnitude the *relationship* between message-passing and computing performance, and also to make the topology of the message-passing network practically invisible. Decreasing the communication latency relative to instruction execution times extends the application span of multicomputers from easily partitioned and distributed problems (*eg*, matrix computations, PDE solvers, finite element analysis, finite difference methods, distant or local field many-body problems, FFTs, ray tracing, distributed simulation of systems composed of loosely coupled physical processes) to computing problems characterized by "high flux" [2] or relatively fine-grain concurrent formulations [3, 4] (*eg*, searching, sorting, concurrent data structures, graph problems, signal processing, image processing, and distributed simulation of systems composed of many tightly coupled physical processes). Such applications place heavy demands on the message-passing network for high bandwidth, low latency, and non-local communication. Decreased message latency also improves the efficiency of the class of applications that have been developed on first-generation systems, and the insensitivity of message latency to process placement simplifies the concurrent formulation of application programs.

Our other goals included a streamlined and easily layered set of message primitives, a node operating system based on a reactive programming model, open interfaces for accelerators and peripheral devices, and node performance improvements that could be achieved economically by using the same technology employed in contemporary workstation computers.

By the autumn of 1986, these ideas had become sufficiently developed, molded together, and tested through simulation to be regarded as a complete architectural design. We were fortunate that the Ametek Computer Research Division was ready and willing to work with us to develop this system as a commercial product. The Ametek Series 2010 multicomputer is the result of this joint effort.

Architecture

Overview

Each Ametek Series 2010 node includes a 25MHz Motorola 68020 processor with a M68881 or M68882 floating-point coprocessor, zero-wait-state memory management hardware, up to 8MB of memory, and a VME interface for accelerators or peripheral controllers. These nodes are about an order of magnitude faster and have about an order of magnitude more memory than those in the first-generation systems. The multicomputer is normally hosted from Sun-3 workstation computers, which also use M68020 processors; hence, the native Sun compilers are able to generate process code for the nodes.

What most distinguishes the Ametek Series 2010 multicomputer from the first-generation "hypercubes" is its message-routing and message-handling hardware. Given our objective not only of keeping pace with the order-of-magnitude advance in node computing performance, but of improving the relationship between communication and computing latencies, we were seeking a major improvement in communication performance.

The way in which this improvement in message performance was achieved was with a combination of organization and technology. The Ametek 2010 does not use a binary n-cube (hypercube) connection network, but instead uses a two-dimensional routing mesh of high-performance custom routing chips. This low-dimension network minimizes latency for a given wire bisection of

the network by allowing more parallel wires and higher bandwidth for each channel. The "wormhole" routing method, unlike store-and-forward routing, does not use storage bandwidth or computing cycles in nodes through which a message is routed. Packets are injected into the network by the source node and leave the network only at the destination node. The entire edge of the mesh is available for hosts or peripheral devices. In order to reduce the software component of the message latency, the nodes include a microprogrammed second processor that manages the send and receive queues.

Communication Network

The Ametek Series 2010 message network is composed of a two-dimensional mesh of custom Mesh Routing Chips (MRCs) [5]. The communication channels are 8 bits wide, and operate self-timed at well in excess of 20MHz, yielding a communication bandwidth per channel of at least 20MB/s (160Mb/s). A higher channel bandwidth is feasible but not economic, since it would exceed even the sequential-access memory bandwidth in the nodes. A node that is sending and receiving concurrently at 20MB/s must on average be performing ten 32-bit accesses per μs.

Message packets advance directly from MRC to MRC in a blocking variant of cut-through routing [6] that we call "wormhole" routing [3,5,7]. The time required to advance the head of a packet from MRC to MRC is only about two byte times. Thus, for example, the time required to send a 64-byte packet (8 double-precision floating-point operands) from corner to corner in a 64-node 8×8 mesh (distance 14) is $0.05(2\times14+64)\mu s = 4.6\mu s$. One may think of this packet as requiring $1.4\mu s$ for path formation and an additional $3.2\mu s$ to spool the message through the channels. For message lengths that are typical of medium-grain multicomputer programs, the length in bytes is considerably larger than the distance in the mesh; hence, the length-dependent component of the latency dominates, and the latency exhibits little sensitivity to message distance.

The performance of this wormhole routing network cannot be compared by a single measure with the performance of the software-controlled store-and-forward packet cut-through message systems in first-generation multicomputers. The store-and-forward networks consume storage bandwidth and computing cycles in the routing nodes, while accumulating a latency of several hundred μs per hop. The case that is most critical for exploiting finer-grain concurrency (*eg*, relatively fewer instructions between message operations, and typically shorter messages) is short non-local messages. The same corner-to-corner message that is delivered in $4.6\mu s$ by the Ametek Series 2010 message network would be handled in a store-and-forward binary 6-cube by the source, destination, and five intermediate nodes, with a total latency of several ms. Thus, in the important case of relatively short non-local messages, the reduction in message latency approaches three orders of magnitude.

The scaling and congestion properties of the network require some comment. In conditions of large applied load to a mesh network, the performance is largely determined by the bisection. Hence, it is desirable to keep the mesh configurations as close to square as possible. A 4×16 64-node machine will function correctly, but has a smaller bisection than an 8×8 configuration. Under an assumption of fixed wire bisection, a two-dimensional network minimizes latency [3,8] for our centerline design point of $N = 256$, a 16×16 mesh. Smaller machines have a surplus of network bandwidth, while larger machines are capable with intense, non-localized message traffic of driving the message network to a state of moderate congestion and consequent noticeable latency. However, according to our simulations, low-dimension networks are very effective in source-queueing packets when the applied load exceeds the network capacity, such that the throughput of the network remains close to its peak operating point.

To realize this scaling in practice, the basic packaging unit in the Ametek Series 2010 is a 4×4 submesh of 16 nodes. The 4×4 submesh is built as an active backplane, measuring 17×12 inches^2, into which the node boards are plugged. These submeshes can be connected vertically and horizontally with other 4×4 submesh units to construct systems up to $32\times16 = 512$ nodes. Still larger systems are perfectly feasible; however, to confine their vertical dimension, they would be constructed with special backplanes with 2×8 or 1×16 submesh units.

Node Architecture

The small network component of the message latency, although important in part for avoiding congestion by sending packets through the network in short bursts, requires equal attention to minimizing the "startup" time or software component of the message latency. The message primitives have accordingly been streamlined so that messages are sent and received from dynamically allocated memory, and the node is an unsymmetrical two-processor architecture. The M68020 and a microprogram-controlled message interface processor share access to main memory and cooperatively maintain data structures consisting of linked control blocks that point to message pages. One structure includes the receive queue and preallocated pages for incoming messages, and the other includes the send queue. Block transfers between memory and hardware queues in the message interface processor are accomplished in static column mode, one of the efficient, high-bandwidth, sequential-access modes of modern dRAM chips. The main memory bandwidth in this mode is a 32-bit cycle each 80ns, or 50MB/s.

Static column mode is also used for the M68020 access, with the most recently accessed column in each 1MB bank serving as a 2KB fast page, similar in effect to a cache set. Thus, a typical 4MB node maintains four fast pages from which the 25MHz M68020 can run with no wait states. Nodes with more memory have proportionately more fast pages. The dRAM refresh is accomplished by hardware. The address translation unit is implemented

with fast static RAMs, with 8KB pages for the code, data, and stack regions, and 256B pages for the dynamically allocated message region. Regions associated with the same process are normally mapped into separate banks so that contiguous code, data, stack, and message references will introduce no wait states.

Messages that are longer than 256B are fragmented into packets with 256B payloads, so that long messages will not block other traffic in the message system for long periods. The size of message pages and the maximum packet length are the same, so that fragmentation and reassembly are accomplished without copying. Taken together, the use of a fast dRAM sequential access mode and the remapping of packets to messages is very effective. Even with the software overhead of fragmenting and reassembling long messages, the asymptotic bandwidth in sending long messages from node to node is higher than the bandwidth that the M68020 achieves copying blocks within the memory of a single node.

The Ametek Series 2010 node design does not compromise in any way with protection. A user process can access only its own data and messages. The node hardware is designed to support not only multiprogramming, but multiple users and virtual memory operation.

Each node also has a high-performance VME interface for peripheral controllers (such as disk interfaces) and accelerators (such as a standard 20Mflop floating-point vector processor).

Programming

The Ametek Series 2010 employs the same process model that was supported on the Cosmic Cubes and first-generation commercial systems. However, in order to streamline the message handling and to allow for efficient layering of a variety of message functions, the primitive message functions are quite different from those used in the first-generation multicomputers. The programming system described here [9] was developed in our research group, and has been in regular use for the past year on the Cosmic Cubes and other multicomputers operated by our group. It was ported to the Ametek Series 2010 without any notable difficulties.

Node Operating System

The standard node operating system for the Ametek 2010 is a proprietary adaptation of a new multicomputer operating system, the *Reactive Kernel* (RK). RK is based on a small kernel that dispatches to kernel processes called *handlers* according to the *tag* in the message at the head of the receive queue. Different handlers and their associated user library routines support different sets of message primitives, as may be required for different languages and applications. Different handlers may be coresident in "subcubes" of the Ametek 2010, so that in the usual *space sharing* mode of operation, different programs can be run concurrently with different primitives. With a suitable handler and library, the Ametek Series 2010 can support the message primitives of any of the first-generation multicomputers.

Host Runtime System

The host runtime system is derived from the Cosmic Environment system (CE version 7.2). The CE system consists of a set of daemon processes, utility programs, and library routines. It handles the allocation of one or more multicomputers, and supports uniform communication between UNIX and node processes. The UNIX processes may all be on the hardware host, or may be distributed among multiple hosts on the same network.

The CE system supports numerous program development features, and is commonly run not only as a host runtime system for multicomputers, but also on single UNIX systems, across networks of UNIX systems, and on multiprocessors. It is a "combat-proven" system that has now been distributed to more than 100 sites. Instructions for obtaining a CE distribution are included in the programming guide [9], which is available from the Caltech Computer Science librarian.

User Programming

The *reactive handler* and its associated library support a set of user interface routines that are analogous to those used in the system interface between a hander and the kernel. Although illustrated here as they are called from processes written in C, user interface routines also exist for other languages.

As usual, each process has a unique identifier consisting of the node number and a process identifier within the node, *viz:* `(node, pid)`. Process spawning is dynamic, and can be initiated from any node or host process with the function:

```
spawn("filename", node, pid, "")
```

Also, as usual, messages are directed to processes, and are queued in transit, but message order is preserved between pairs of communicating processes. Within the limits of the computation being deterministic and not exceeding available storage sizes, the results of a computation do not depend on the way in which the processes are distributed.

Messages are sent and received from dynamically allocated memory that is accessed both by user processes and by the message system. Message buffers are arrays of bytes with no presumed structure, and the C functions that return pointers to message buffers return maximally aligned pointers of type `char*`. Message space can be allocated by:

```
p = xmalloc(length);
```

where the `length` of the block pointed to by p is specified in bytes, and can be deallocated by:

```
xfree(p);
```

These functions are semantically identical to the usual UNIX `malloc` and `free` functions.

When a message has been built in a block that has been allocated, its contents can be sent as a message by:

```
xsend(p, node, pid);
```

The xsend function also deallocates the message block; that is, `xsend(p,...)` is like `xfree(p)`, except that it also sends a message. Thus, there is no need for blocking or for feedback that the message has been sent. When the function returns, the message block is gone.

Messages can be received by:

```
p = xrecvb();
```

such that p then points to a new message. As indicated by the "b" at the end of xrecvb, this is a *blocking* function that does not return until a message has arrived for the process.

The execution of the xrecvb function is just like allocating a message buffer with xmalloc, except that the length of the block allocated is determined by the length of the message received. Once the message contents are no longer needed, the allocated space should be freed. Of course, the message space can be freed with `xfree(p)`, but it can also be freed by `xsend(p, ...)` if there is a message of the same length to send. It frequently happens in message-passing programs that a message that is received is simply modified by a computation and then sent on to another process.

The non-blocking receive function is called xrecv. It is required only for applications in which a process may need to *probe* for another message without giving up the right to continue execution. The usage of the xrecv function is identical to xrecvb; however, it may return a NULL pointer if there is no message queued for this process. This behavior of the xrecv function allows one to write programs that can do other work while waiting for a message; for example:

```
while (1) {
    if (p = xrecv()) digest(p);
    else do_other_work();
}
```

In such usage, the digest(p) and do_other_work() functions should return after a bounded time to call xrecv again, because calling xrecv or xrecvb when the next message in the node's receive queue is for another process allows the kernel to save the state of that process and start running the other process. The appearance of xrecv or xrecvb in the code marks a *choice point* for switching the execution to another process, and it is in this sense that the scheduling is *reactive* or *message driven*.

Programs may use these primitive functions directly, or may use other classes of functions that are expressed in terms of the "x" primitives. Extra information, such as a message type, can be inserted into extra space allocated in a message buffer, and sent with a message. The function used to receive typed messages can filter them into separate queues of pointers (the messages themselves remain intact) according to the type. For example, the user interface functions defined for FORTRAN allow processes to exercise discretion in the messages received according to any combination of message type and sender ID.

Conclusion

Taken together, the computing and communication performance, scalability, open interfaces, I/O capability, new features, and system software of this second-generation multicomputer represent to us the fulfillment of an "IOU" — a working demonstration of the capabilities we have said would be possible to include in a well-engineered multicomputer.

Acknowledgments

The research that led to the architectural design and system software of the Ametek Series 2010 was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; and in part by a grant from Ametek Computer Research Division.

We very much appreciate the dedicated efforts and support of the employees and management of Ametek.

Certain of the techniques described here are the subjects of patents filed by Caltech and by Ametek. The Cosmic Environment and Reactive Kernel are the property of Caltech, and are licensed to Ametek.

References

[1] Charles L Seitz, "The Cosmic Cube," *CACM*, 28(1): 22-33, January 1985.

[2] J D Ullman, "Flux, Sorting, and Supercomputer Organization for AI Applications," *J of Parallel and Distributed Computing* 1: 133-151, 1984.

[3] William J Dally, *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, 1987.

[4] William C Athas, "Fine Grain Concurrent Computations," Caltech Computer Science technical report (PhD thesis) 5242:TR:87.

[5] Charles M Flaig, "VLSI Mesh Routing Systems," Caltech Computer Science technical report (MS thesis) 5241:TR:87.

[6] P Kermani and L Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," *Computer Networks* 3: 267-286, 1979.

[7] William J Dally, Charles L Seitz, "The Torus Routing Chip," *Distributed Computing* 1(4): 187-196, Springer International, 1986.

[8] William J Dally, "Wire-Efficient VLSI Multiprocessor Communication Networks," *Proc 1987 Stanford Conference on Advanced Research in VLSI*, MIT Press, 1987.

[9] Charles L Seitz, Jakov Seizovic, Wen-King Su, "The C Programmer's Abbreviated Guide to Multicomputer Programming," Caltech-CS-TR-88-1, 1988.